SGX-Bundler: speeding up enclave transitions for IO-intensive applications

Svenningsson, Jakob; Paladi, Nicolae; Vahidi, Arash

Published in:
Proceedings of the 22nd IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing

DOI:
10.1109/CCGrid54584.2022.00036

2022

Document Version:
Early version, also known as pre-print

SGX-Bundler: speeding up enclave transitions for IO-intensive applications

Jakob Svenningsson
KTH Royal Institute of Technology
Stockholm, Sweden
jaksve@kth.se

Nicolae Paladi
Lund University and CanaryBit AB
Lund, Sweden
nicolae.paladi@eit.lth.se

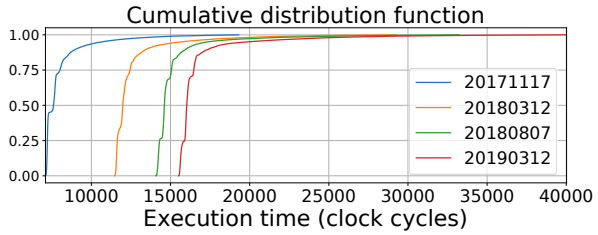
Arash Vahidi
RISE Cybersecurity
Lund, Sweden
arash.vahidi@ri.se

***Abstract*—Process-based confidential computing enclaves such as Intel SGX can be used to protect the confidentiality and integrity of workloads, without the overhead of virtualisation. However, they introduce a notable performance overhead, especially when it comes to transitions in and out of the enclave context. Such overhead makes the use of enclaves impractical for running IO-intensive applications, such as network packet processing or biological sequence analysis. We build on earlier approaches to improve the IO performance of work-loads in Intel SGX enclaves and propose the SGX-Bundler library, which helps reduce the cost of both individual single enclave transitions well as of the total number of enclave transitions in trusted applications running in Intel SGX enclaves. We describe the implementation of the SGX-Bundler library, evaluate its performance and demonstrate its practicality using the case study of Open vSwitch, a widely used software switch implementation.**

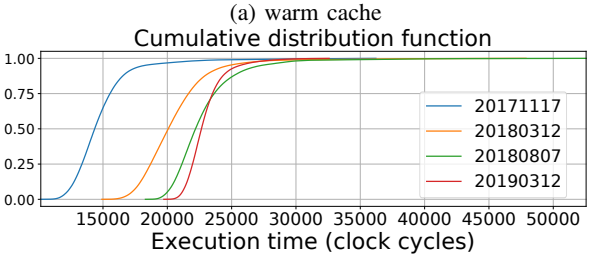
***Index Terms*—SGX, Hardware security, Open vSwitch, Performance optimization**

I. Introduction

Confidentiality and integrity are important topics when computation moves from local premises to a third-party environment. Addressing these topics should not offset the two core advantages of cloud computing: cost reduction and flexibility. *Confidential computing* is an increasingly popular approach to achieving this [35]. It relies on using a Trusted Execution Environment (TEE) backed by certified hardware, such that critical operations of Trusted Applications running inside the TEE cannot be manipulated by the platform operator or malicious entities (with the notable exception of the CPU manufacturer). For example, AMD SEV, Intel SGX, and IBM SVM provide mechanisms to achieve this in different ways [12], [20], [34]. The variety of vendor TEE implementations highlights trade-offs between security guarantees, portability of legacy applications, ease of deployment, and run-time performance. VM-based TEE implementations (e.g. AMD SEV, IBM SVM, and Intel TDX) support portability of legacy applications with a modest performance overhead [8], but have a larger attack surface and are vulnerable to several classes of attacks [17]. Process-based TEEs (e.g. Intel SGX and ARM TrustZone) on the other hand have a smaller attack surface and improved security. Unfortunately, the additional security checks together with memory access limitations also affect the performance of process-based TEEs negatively [8]. Furthermore, these have shown to be particularly vulnerable to microarchitectural attacks [25] and platform vendors have repeatedly issued microcode patches to alleviate security problems [7]. Figure 1 illustrates that microcode updates for Intel SGX have caused TEE performance to decrease even further.



(a) warm cache



(b) Cold cache

Fig. 1: Evolution of the SGX enclave transition time though Intel microcode updates, presented as a cumulative distribution function (CDF).

It is therefore imperative to identify and implement new approaches that help to maintain or improve TEE performance despite the latest countermeasures to microarchitectural attacks. This, however, must also be done in a way that does not significantly increase application developers' efforts. In this paper, we address a crucial limitation on the intersection between the portability of legacy applications and the performance overhead introduced by the transition between the TEE and the Rich Execution Environment (REE).

Our results show that while a tailor-made refactoring of legacy Trusted Applications for Intel SGX yields the best performance, it is labor-intensive, application-specific, and often impractical. This insight led us to develop the *SGX-Bundler* software library as a generic approach for speeding up enclave transitions in legacy Trusted Applications, while maintaining the security benefits of SGX. This solution is particularly beneficial in IO-intensive applications such as network packet

processing, remote sensing applications, biological sequence analysis, and long-running simulations [24]. We demonstrate the practical applicability and performance improvements of our approach using a packet processing application. Note however that the proposed solution is generic and can be applied to any domain. This paper extends, clarifies and complements the preliminary results presented in [28]. The main **contributions** of our work are summarized as follows:

- We describe a generic approach to speed up transitions between the rich execution environment and SGX enclaves (Section III);
- We introduce *enclave execution graphs*, that allow executing arbitrary sequences of enclave functions using a single enclave transition (Section IV);
- We implement a library to assist refactoring of legacy applications and introduce efficient transitions in and out of SGX enclaves;
- We demonstrate the applicability of our approach with the case study of a widely used IO-intensive application;
- The implementation source code is openly available[1].

The rest of the paper is structured as follows. We introduce the required background and motivate the problem in Section II, introduce the SGX-Bundler library in Section III and describe the implementation of the library in Section IV. Next, we evaluate the performance of the SGX-Bundler library and its application in a case study in Section V, present the related work in Section VI followed by conclusion and future work in Section VII.

II. Background

We next introduce several key concepts used in the paper.

A. Intel SGX

Intel Software Guard Extensions (SGX) are CPU security extensions that allow execution of unprivileged trusted applications in the presence of possibly malicious privileged software such as a compromised OS or hypervisor [20]. An SGX-enabled CPU maintains an isolated memory region, the Enclave Page Cache (EPC), within which security enclaves can execute isolated from the rest of the system. SGX provides mechanisms to verify the integrity of an enclave (using local and remote attestation) and binding of information to specific configurations (sealing), which allows one to validate an enclave without direct access to its content.

Enclaves communicate with applications running in the Rich Execution Environment (REE) using the ECALL and OCALL (entry and out call) instructions. However, these instructions introduce a performance overhead that often makes SGX unsuitable for IO-intensive applications. Weisse proposed using a shared memory region outside the enclave for communication, resulting in significant performance improvements in real-world applications [33]. In response to published security vulnerabilities affecting Intel SGX [31], [18], [15], Intel issued a number of microcode updates. However, along with addressing software vulnerabilities this further degraded the performance of enclave transitions (see Figure 1). While the HotCalls approach [33] produces a tangible performance improvement, we note the importance of further efforts to offset the overhead introduced by subsequent microcode updates.

[1]Source code repository: https://github.com/nicopal/sgx_bundler

B. Memoisation

Memoization is an optimization technique for reducing the execution time of computationally expensive functions [14]. Given a function with no side effects, memoization uses a cache to remember some input-output pairs. If an input used in a subsequent call is found in the cache, the recorded output value is returned, otherwise, the (expensive) function call is taken. Memoization is a simple way of trading execution time for space and is commonly used to optimize recursive algorithms. We use memoization to speed up enclave transition times between applications running in the TEE and the REE.

C. Open vSwitch

The motivating use case for this work is Open VSwitch (OvS), a software network switch for connecting physical and virtual network interfaces in a virtualized environment [16]. This is a critical component for providing network isolation in cloud infrastructure and other multi-tenant environment [23].

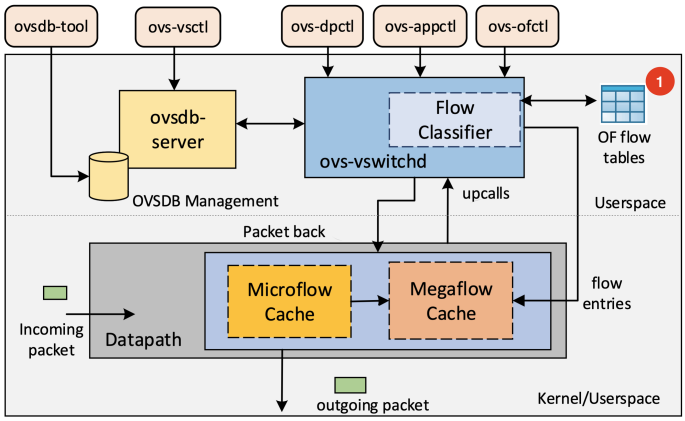


Fig. 2: Overview of OvS main components.

Among the OvS components (Figure 2), the *flow tables* (1) are of special interest to us as they contain the rules that define the switch routing behavior. While these tables are critical OvS assets, they are often stored without sufficient confidentiality and integrity protection, leading to serious security vulnerabilities. For example, an attacker with access to flow tables could map the network structure [5], modify routing behavior to perform man-in-the-middle attacks, or bypass firewalls and intrusion-detection systems [4]. Furthermore, an attacker could inject malicious data into flow tables to propagate deeper into the network and compromise systems otherwise not reachable [9].

Proposed solutions to address flow table security issues include auditing flow table to detect discrepancies between

the configured and current behavior [19], validating both executables and flow tables with a TPM [11], or moving critical components (the OpenFlow flow tables and forwarding logic) into Intel SGX enclaves [21]. The latter, while promising from a security point of view, is a very labor-intensive task and introduces additional overhead. In this work, we address both shortcomings.

D. Threat model

We focus on the integrity of critical components in applications executing in multi-tenant environments. We assume the critical components execute in TEEs and communicate at high frequency with the corresponding applications in REE. We consider an adversary capable of operating arbitrary software components and having remote execution capabilities on platforms where target applications operate. The adversary may modify any REE software component. We exclude microarchitectural attacks [7] and address them in upcoming work; we consider existing countermeasures against such attacks in our performance analysis.

III. Speeding up Enclave Transitions

We describe *SGX-Bundler*, a mechanism addressing performance penalties caused by transitions from and to SGX enclaves. To help adoption and usability, we implemented the *SGX-Bundler* library.

A. Overview

The *SGX-Bundler* library offers functionality to reduce the cost of individual enclave transition as well as the total number of enclave transitions for trusted applications (TAs) deployed in Intel SGX enclaves. This library extends work conducted in HotCalls [33] with novel ideas and is the core contribution of this paper. The library leverages three main features: switchless enclave function calls, execution graphs, and enclave function memoization.

Switchless enclave function calls are used to reduce the cost of a single enclave transition. Execution graphs and enclave function memoization are used to reduce the total number of enclave function calls in Intel SGX applications.

B. Functional Requirements

We consider the following functional requirements for the SGX-Bundler library, defined based on observations of the performance analysis described in Section V-C: (1) **Switchless calls**: execute enclave functions without context-switching to enclave mode; (2) **Merging**: execute an arbitrary number of enclave functions over a single enclave transition; (3) **Batching**: apply an arbitrary number of enclave functions to each element of an input list over a single enclave transition; (4) **Branching**: conditional execution of enclave functions over a single enclave transition; (5) **Memoization**: cache enclave data in untrusted memory when confidentially is not required. Caches allow untrusted applications data access without enclave transitions. Moreover, we implement a mechanism to verify the integrity of enclave data stored in untrusted memory.

The switchless enclave function call component presented in IV-A fullfills requirement 1; the execution graph component in Section IV-B fullfills requirements 2-4, and the memoization component in Section IV-D fullfills requirement 5.

C. Architecture

In the case of SGX enclaves, implementing a shared memory switchless enclave communication library requires source code modifications in both the trusted application running in the TEE and the untrusted application running in the REE. Enclaves do not share source code (and libraries) with the untrusted application; therefore, the SGX-Bundler library consists of two separate libraries. The first library is a *static C* library that needs to be linked with the untrusted application, and the second is a trusted enclave library which needs to be linked with the enclave. Trusted enclave libraries are static libraries that are linked with the enclave binary [1].

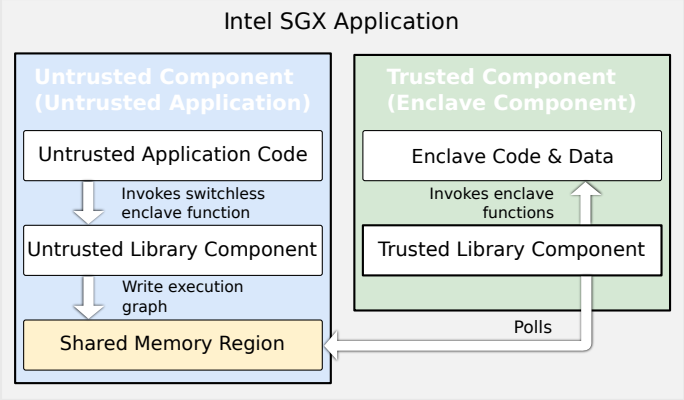


Fig. 3: High-level overview of an Intel SGX application using the SGX-Bundler library.

Figure 3 illustrates the untrusted and trusted part of the SGX-Bundler library when integrated into an arbitrary Intel SGX application and the interactions between the different parts. The untrusted application invokes switchless enclave functions through an API exposed by the untrusted library. Next, the untrusted library writes the job to a shared memory region in the form of an execution graph (execution graphs are discussed later in Section IV-B). Finally, the job is processed by an enclave worker thread which calls the associated enclave function and writes back potential return values to the shared memory region.

IV. SGX-Bundler Implementation

We next describe the SGX-Bundler implementation.

A. Switchless Enclave Function Calls

The protocol used for switchless enclave function calls in the SGX-Bundler library builds on HotCalls [33] and is presented in Figure 4. This component fulfills functional requirement (1) listed above in III-B. The shared memory region contains a *spinlock* primitive that must be acquired by

either the untrusted application and the TA before accessing the shared memory region to avoid data races. While Intel SGX SDK supports condition variables, this synchronization primitive is implemented with OCALLS, which is a context switch operation and conflicts with our goal to keep the communication protocol switchless. Spinlock is the only synchronization primitive that can be used by the enclave worker threads without leaving the enclave.

The untrusted application invokes switchless enclave functions by acquiring the lock of the shared memory region and writing the enclave function call, represented by a (*function_id*, *function_data*) tuple to shared memory. An enclave worker thread initiated through an API exposed by the trusted part of the library is continuously polling the shared memory region for scheduled jobs to execute. The enclave worker thread uses a busy-waiting scheme where it repeatedly checks for pending jobs inside of an infinite loop. We use Intel's *pause* instruction inside of the spinlock loop to improve the efficiency of the busy-waiting scheme. The pause instruction provides a hint to the processor that it is executing inside a spinlock loop, enabling the processor to perform memory optimizations [10].

In Section IV-B we will replace this tuple with a data structure representing an *execution graph* to create a more efficient enclave communication scheme able to execute multiple enclave functions using a single enclave transition.

1) Translation Functions: Input and output parameters are treated as generic elements which simplifies the implementation but must be translated to correct data types before an enclave worker thread can be invoked. This is done by defining a translation function for each function exposed to the untrusted application, see Listing 1 for an example. Note that translation functions are constructed to accept an array of parameters, which will enable the use of batching (see Section **??**).

Listing 1: A translation function for an enclave summation.

```
void translation_ecall_plus( unsigned int itrs,
  unsigned int params, void *args[][]) {
  for(int i = 0; i < iters; ++i) {
    *(int *) args[2][i] = hotcall_plus(
        *(int *) args[0][i], *(int *) args[1][i]);
  }
}
```

B. Execution Graphs

A limitation of the *HotCall* implementation [33] is that it only allows execution of a single enclave function per enclave transition. A switchless enclave transition still introduces an overhead estimated to be around ~600 to ~1400 clock cycles for warm and cold caches respectively [33]. A simple approach to address this is to merge sequence of enclave calls into a single call, as illustrated in Figure 5.

In practice the enclave call sequence may be much more complex than just a pre-defined list of function calls. To address this, we introduce the concept of *execution graphs* in the context of enclave transitions. An enclave *execution graph* is an arbitrary sequence of dependent or independent enclave function calls, control statements, and iterators that are executed within a single enclave transition. This provides a significant improvement over the original HotCall implementation and is to best of our knowledge a novel concept that has not been explored in previous studies. We discuss various graph components and their function outlines in our Technical Report [29].

C. Construction of Execution Graphs

When converting an imperative programming language to execution graphs, each node can require $5 - 10$ lines of boilerplate code. This is a tedious and error-prone task and most likely will result in a less readable code. To address this problem, we created a user-friendly API based on C preprocessor macros. This API can be used for building execution graphs using both an imperative and functional-style syntax, and is explained in detail in the technical report accompanying this paper [29].

D. Enclave Function Memoization

While execution graphs are effective capturing complex operations that have a high number of enclave calls, they are not as effective in handling simpler enclave operations. To address this we make propose caching results of frequent enclave calls in untrusted shared memory using a technique called *memoization*. The integrity of memoization caches in untrusted memory is guaranteed by storing a hash of each memoization cache in the enclave. We compute the hash of a memoization cache as follows:

$$\sum_{e \in C} hash(e) \tag{1}$$

where C is the set of all entries in the cache. The enclave worker thread, responsible for populating memoization caches, updates the corresponding memoization hash each time a cache entry is inserted or deleted.

The enclave worker thread periodically verifies the caches by recalculating the hashes of the memoization caches in untrusted memory and compares them with the hashes stored in enclave memory. Depending on the nature of the application, different actions can be appropriate when an unauthorized modification is detected.

Manipulating the eviction list only enables an attacker to give cache priority to selected entries.

V. Results

To assess the performance gains of using SGX-Bundler, we first perform several micro-benchmarks for each component of the library. To evaluate real-world performance improvements in a significantly more complex environment, we also evaluate a prototype implementation of OvS with SGX support.

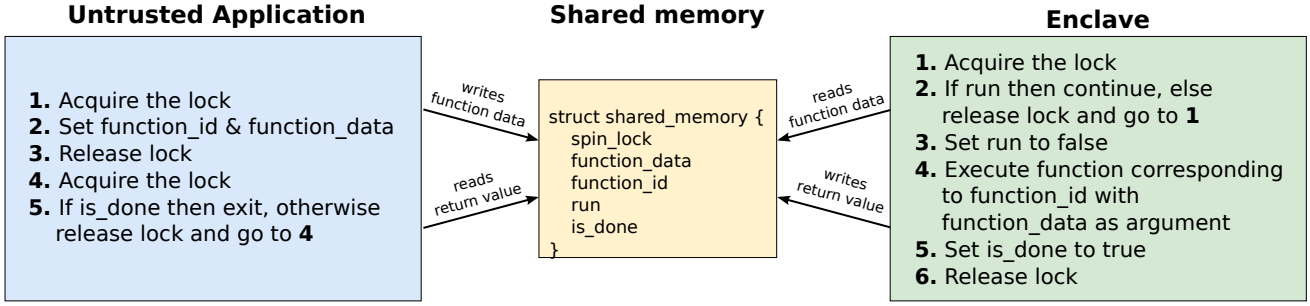
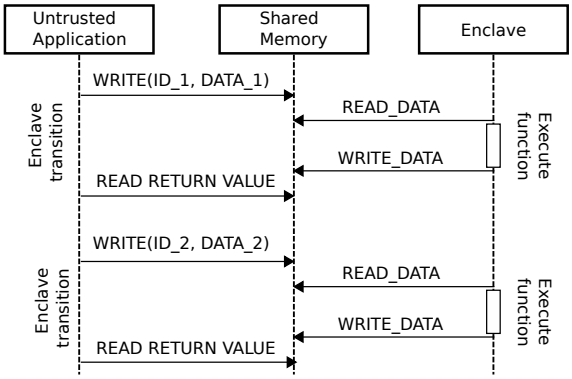
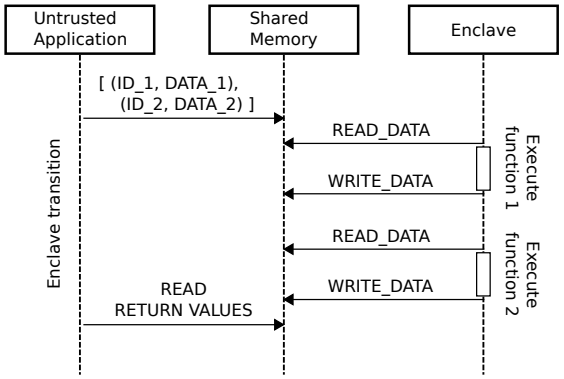


Fig. 4: Switchless enclave function call protocol.

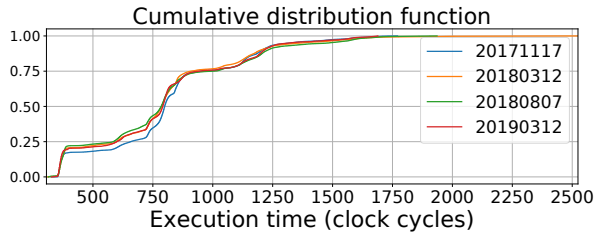


(a) Two enclave calls

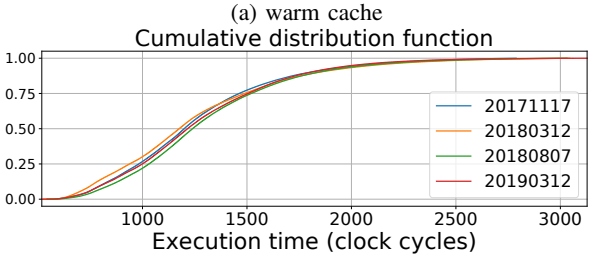


(b) Two merged enclave calls

Fig. 5: Sequence diagram illustrating two function calls with the original HotCall implementation and using execution graphs.



(a) warm cache



(b) cold cache

Fig. 6: Enclave transition times for switchless enclave function calls with different Intel microcode versions.

A. SGX-Bundler Library

The following components of the proposed solution are evaluated here: switchless enclave functions, execution graphs (merging, batching, branching), and memoization. This allows us to study the benefits of each improvement in isolation.

1) Enclave Transition Time: Measured execution time for switchless function calls can be observed in Figure 6. Compared to ECALLs (see Figure 1), we note that not only the switchless calls are significantly faster (~**20.3x** and ~**18.6x** for warm and cold cache), they are also mostly unaffected by microcode updates. In contrast, ECALL warm and cold cache performance has increased over time by ~110.8% and ~57.8% respectively.

B. Execution Graphs

We next evaluate the merging, batching and branching capabilities of the SGX-Bundler library.

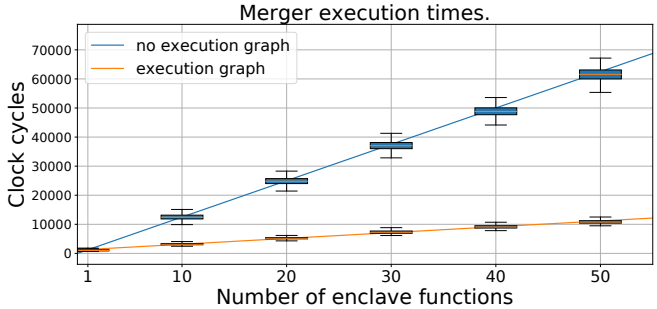
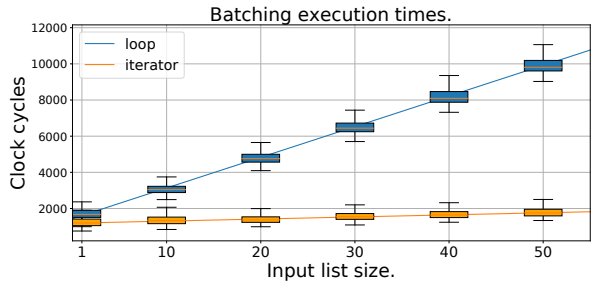


Fig. 7: Execution times when executing n different enclave functions, with and without execution graphs.

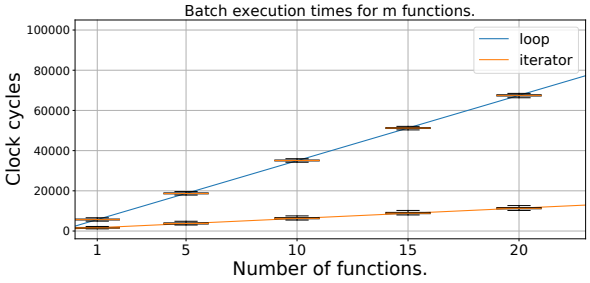
1) Merging: We compared execution time of merging $n \in \{1, 10, 20, 30, 40, 50\}$ enclave function calls using execution graphs to single enclave function calls (i.e. no merging). As Figure 7 illustrates, merging significantly reduces execution time when multiple calls are performed. Note that without execution graphs the execution time is dominated by the transition time measured in Figure 6.

2) Batching: Given a list of $n \in \{1, 10, 20, 30, 40, 50\}$ elements, we compared the processing time using either a loop or an iterator. As Figure 8a illustrates, iterators are ~14x faster even though both operations grow linearly with the number of elements.

Currently, iterators are limited to a single enclave call per round while loops can perform an arbitrary number. We measured the execution time of applying $m \in \{1, 5, 10, 15, 20\}$ enclave functions to an input list of size 20. Despite iterating through the list m times, iterators are still ~6.4 times faster than loops (see Figure 8b).



(a) One function, n elements.

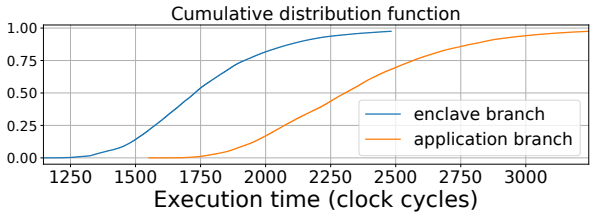


(b) m functions, 20 elements.

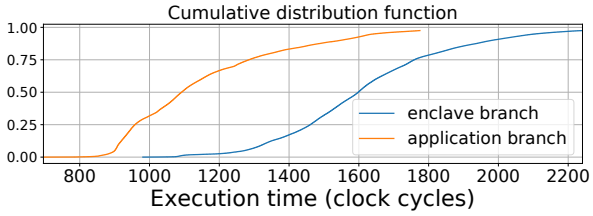
Fig. 8: Execution time when applying m enclave functions to each element in a list of size n.

3) Branching: In Figure 9 we compare enclave branching, which happens inside the enclave to application branching which performs the branch in the application and executes the branch body as a separate execution graph. Notice that enclave branching is faster when the branch condition is true, but slower when false. However, if the branch operation is repeated at least once (Figure 9 (c)), then enclave branching is faster even when the condition is false.

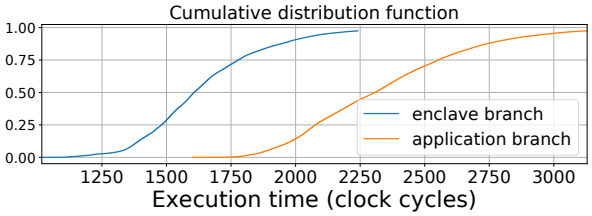
4) Enclave Function Memoization: We measure performance gains of memoizing a variable accessed through an enclave function. In Figure 10 we measure execution time for



(a) True condition



(b) False condition



(c) False condition in loop with 2 iterations

Fig. 9: Execution times for enclave and application branching.

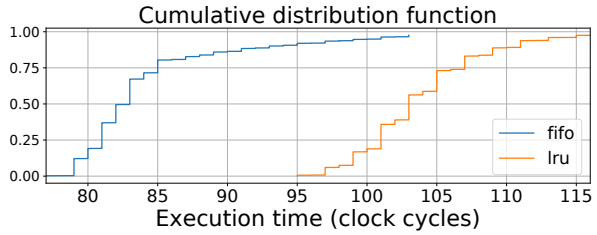
a cache hit with LRU or FIFO eviction policies. Notice that a cache hit is ~20-24x faster than a cache miss.

At the same time, memoization introduces a noticeable overhead to enclave operations that update a value and thus must update or invalidate the cached value. Figure 11 shows that the cost differs depending on the state of the cache (~12.0% and ~20.7% for warm and cold caches respectively).

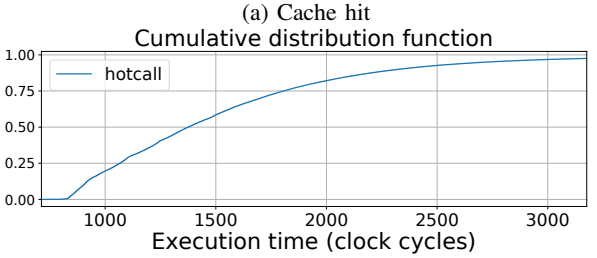
C. OvS Prototypes

To better assess the proposed solutions it is important to study the performance impact in non-trivial real-world applications. Since time constraints would limit us to analysis of a single application, we chose Open vSwitch (commit `53cc4b0`) where different operations and workloads should cover a wide range of executions characteristics. To this aim, we studied four Open vSwitch flow table operations (add, delete, modify and evict flow rule) under realistic SDN workloads and as average of 20 separate rounds.

The performance of each operation has been compared across five different implementations: *baseline* is the original version, *SGX vanilla* is the OFTinSGX version [21], *Switchless* uses hotcalls instead of ECALLs [33] while *Bundler* uses all optimizations described in this paper. Finally *SGX refactored* is the authors heavily modified version tailored specifically



(a) Cache hit



(b) Cache miss

Fig. 10: Execution times of an enclave function with memoization enabled for both cache hits and misses.

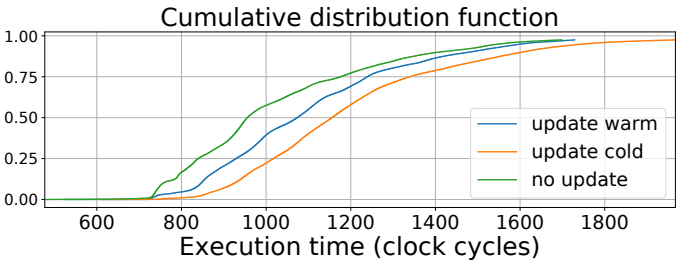


Fig. 11: Enclave function execution times with and without cache updates.

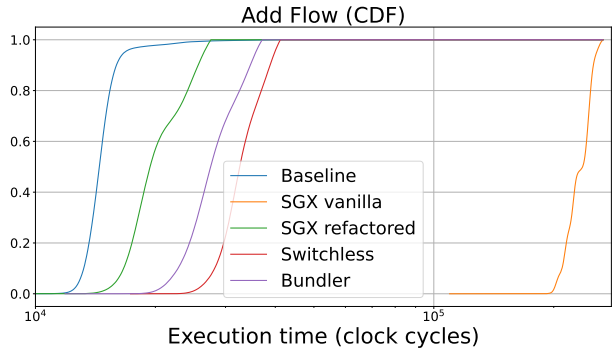
for SGX and will be used to compare the trade-offs between performance and development effort. All evaluation scripts are openly available[2].

1) Add and Delete Flow Rules: Measured execution time for add and delete flow operations can be seen in Figure 12 and Table I. We note that in both cases *SGX vanilla* performs significantly worse than other implementations. It is also noted that *Bundler* performs slightly better than *Switchless* but not as well as *SGX refactored* which is the best preforming secure implementation (with *Baseline* being the best performing overall implementation.)

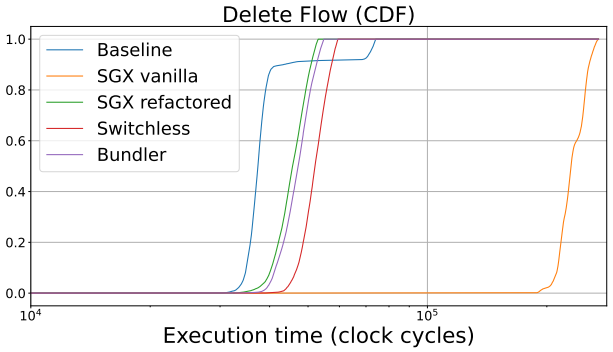
The difference between *Bundler* and *Switchless* increases slightly in the case of delete operations. Unlike add, delete operations often target multiple table entries and therefore benefits from batching. We will shortly revisit this difference for other operations

2) Modify and Evict Flow Rules: Measured execution time for modify and evict operations can be seen in Figure 13 and

[2]Source code repository: https://anonymous.4open.science/r/sgx_bundler-E9C2



(a) Add flow



(b) Delete flow

Fig. 12: Execution time for add and delete flow operations

Table I. The pattern observed earlier is repeated here, although the performance difference between *Bundler* and *Switchless* is growing. For example, for evict operations the former is well over an order of magnitude faster.

We note that modify and evict represent more complex operations that may require many more enclave transitions. In such situations *Bundler* appears to provide much more stable performance improvements. Hence we conclude that while *Switchless* provides some performance improvements, its benefits are limited in IO-intensive applications.

Given these results, it seems that while *Bundler* introduce a measurable performance overhead it does not drastically increase execution time even in corner cases. We note that *SGX refactored* demonstrates better performance, but is the performance gain worth the amount of work required to rewrite the original application?

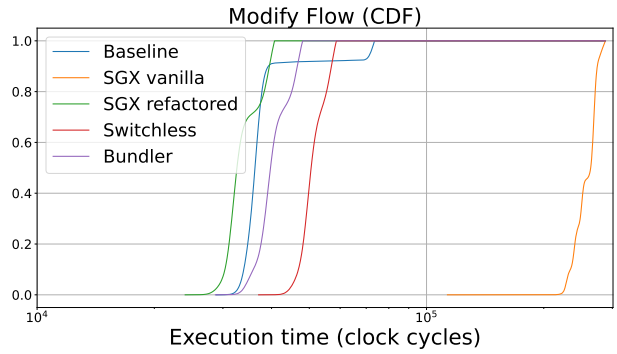
D. Programming Effort Trade-Offs

The effort required to rewrite and optimize an application specifically for SGX depends on the size and complexity of the application. In practice doing this may not be possible due to cost and time limitations, lack of know-how or other issues. For example, maintainability may suffer as transferring new changes from the original project to the rewritten version becomes much harder. As mainline changes also include security patches, this might also negatively affect the security.

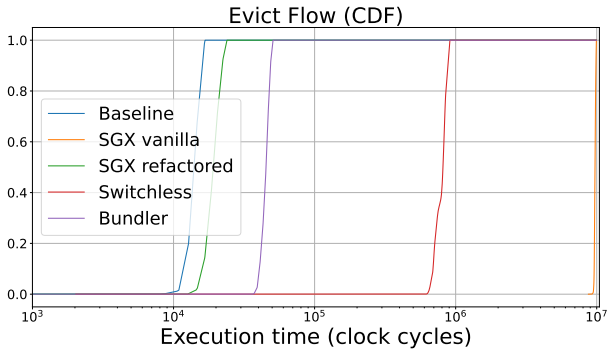
While in this work the authors were able to create the *SGX refactored* implementation for Open vSwitch, the effort

TABLE I: OvS overhead for different operations

Version	Batch size	Op	Overhead for quantile 25%	50%	75%	Op	Overhead for quantile 25%	50%	75%
Baseline	-	Add	0	0	0	Delete	0	0	0
SGX Refactor	-	Add	29	34	53	Delete	19	22	26
Bundler	1	Add	80	90	106	Delete	22	26	29
Bundler	16	-	-	-	-	Delete	17	16	15
Switchless	-	Add	119	124	136	Delete	37	39	42
SGX vanilla	-	Add	1473	1524	1511	Delete	508	517	544
Baseline	-	Modify	0	0	0	Evict	0	0	0
SGX Refactor	-	Modify	-11	-10	0	Evict	23	25	27
Bundler	1	Modify	8	10	16	Evict	153	153	150
Bundler	16	Modify	21	21	21	Evict	86	89	91
Switchless	-	Modify	38	39	45	Evict	3661	3906	3860
SGX vanilla	-	Modify	588	622	615	Evict	49628	47606	45703



(a) Modify flow



(b) Evict flow

Fig. 13: Execution time for modify and evict flow operations

for doing so was not negligible. In comparison the *Bundler* implementation utilized the SGX-Bundler library and required much smaller changes to the Open vSwitch source code (less than 1% of lines and 1% of files were modified). We believe this is mainly attributed to the user API for constructing execution graphs (see the Technical Report [29]).

E. Security Analysis and Limitations

We next assess the security implications of using SGX-Bundler by examining the changes that can affect the trusted code running inside the enclave.

Storing execution graphs in shared memory does not advantage an attacker in the Intel SGX adversary model, where the attacker controls the underlying OS and the enclave IO [3]. SGX-Bundler applies the approach employed by Hotcalls [33] and SGX SDK (reference) for passing the data structures between the untrusted code and the enclave. Furthermore, execution graphs are initially constructed in the untrusted, rich execution environment and are not sensitive to attacks on confidentiality and integrity (beyond performance effects). We further refer to the security analysis in [33] which analyzes a similar approach.

Replay attacks and Denial of Service: The SGX-Bundler approach does not introduce additional risks for replay attacks or denial of service. Calls to the API of the trusted application running in the enclave are always issued from the untrusted rich execution environment. Denial of service is outside the Intel SGX adversary model.

Confidentiality of application data: SGX-Bundler does not affect data confidentiality as neither user nor application data is stored in the shared memory. We assume that all user data communicated between the trusted enclave and the untrusted rich execution environment is done through a secure channel established following enclave attestation [3].

Integrity of the memoization cache: An attacker can temporarily change the content of a memoization cache without being detected. If an attacker modifies a memoization cache entry and restores the original value before the next memoization cache verification, then the unauthorized modification will not be detected by the enclave. An attacker can time the cache changes such that cache modifications are removed before each cache verification, thus hiding the changes. However, assuming the attacker cannot determine exactly when the enclave worker thread verifies memoization caches, the integrity violation will be eventually detected. As a result, enclave function memoization only guarantees *eventual integrity* of its content and is not directly suitable for settings requiring stronger integrity guarantees. Instead, it is appropriate in contexts when the logic of the application must be protected, and not the data that it processes. Consider the case of proprietary algorithms, software implementations, parameters of DNN models, and other types of software intellectual property.

VI. Related Work

Performance issues in SGX applications can sometimes be attributed to the high cost of entering and exiting enclaves. Weisse *et al*. introduced "*HotCalls*" for communicating with enclaves using shared untrusted memory [33]. This approach can be orders of magnitude faster than ECALLs, although the use of untrusted memory also increases the attack surface for the enclave. The switchless enclave function call component of the *SGX-Bundler* library developed in this paper, presented in Section III-A, is heavily inspired by this work.

The HotCalls protocol requires an enclave worker thread that communicates with the main thread through a shared memory region. This thread will occupy one CPU core, which is economical only when the SGX enclave is under some load. Tian *et al*. suggested using an adaptive approach where ECALLs are used when the device is mostly idle and switchless calls are used when it is under some load [30]. This scheme has been included in recent versions of the Intel SGX SDK as an official feature. We chose to not use this scheme in our paper as it lacked the flexibility and control granularity of a custom solution.

ShieldStore is an SGX enabled key-value store that uses HotCalls and also overcomes EPC memory limitations by storing all key-value pairs in encrypted untrusted memory [13]. The two prototypes developed in this work were highly influenced by ShieldStore.

The authors of [6] presented an extensive performance study for virtualization and Intel SGX. The study includes a large number of benchmarks on ECALL, OCALL, and EPC paging performance in native and virtual environments. The native ECALL performance estimates were used in Section V-C of this paper.

Weichbrodt *et al*. presented *sgx-perf*, a performance analysis tool for Intel SGX applications [32]. Using this tool, the authors analyzed scenarios where Intel SGX was a significant performance bottleneck and suggested possible solutions. Two such scenarios were subsequent calls of the same enclave function and subsequent calls to different enclave functions. While the authors proposed batching and merging respectively or moving the caller into the enclave, to the best of our knowledge this has not been implemented and described before our work. Furthermore, we improve the usability of this approach by packaging it as a library.

Software-Defined Networking (SDN) and in particular the SDN control plane has been extensively scrutinized by security researchers [27], [2]. Some researchers have considered the use of Trusted Computing and trusted execution to address security issues. For example, Jacquin *et al*. proposed using TPM to ensure a trusted boot and use of attestation to monitor the integrity of flow tables [11], while Paladi *et al*. suggested using Intel SGX to ensure a secure boot and to provide secure communication channels [22]. Similarly, Shih *et al*. proposed executing parts of a virtual network function inside an Intel SGX enclave [26].

Medina *et al*. proposed *OFTinSGX*, an Open vSwitch implementation where OpenFlow flow tables are placed inside an SGX enclave [21]. While this provided confidentiality and integrity guarantees to the flow tables, it also brought a significant performance degradation to OvS.

VII. Conclusions

In this paper we presented SGX-Bundler, a mechanism to help improve the performance of IO-intensive applications in Intel SGX enclaves. The proposed mechanism combines switchless SGX communication and a novel optimization using execution graphs and function memoization. We extended earlier work and developed two prototypes that utilize switchless communication both with and without execution graphs and memoization. Our evaluation shows that while switchless communication contributes to some performance improvements, the addition of execution graphs and memoization leads to further significant improvements. In particular, our approach seems to be much better equipped to handle exceedingly IO-intensive operations, making it more suitable for real-world usage. The suggested improvements come however at the cost of increased size and complexity. To facilitate adoption we encapsulated the proposed mechanism into the openly available SGX-Bundler library, thereby reducing development effort significantly.

We thoroughly evaluated the performance improvements introduced by the SGX-Bundler library using the case study of Open vSwitch, a widely used network switch implementation. The SGX-Bundler library can be used for other IO-intensive applications that can benefit from the security guarantees of isolated execution in SGX, such as biological sequence analysis or long-running simulations. Considering the many parameters that each evaluation entails, we will explore in future work the performance effects of the SGX-Bundler library in further applications, as well as evaluate the required programming efforts across several case studies.

References

[1] Intel Software Guard Extensions Programming Reference. Tech. rep., https://01.org/sites/default/files/documentation/intel_sgx_sdk_developer_reference_for_linux_os_pdf.pdf

[2] Abdou, A., van Oorschot, P.C., Wan, T.: Comparative Analysis of Control Plane Security of SDN and Conventional Networks. IEEE Communications Surveys&Tutorials **20**(4), 3542–3559 (2018). https://doi.org/10.1109/COMST.2018.2839348

[3] Anati, I., Gueron, S., Johnson, S., Scarlata, V.: Innovative technology for CPU based attestation and sealing. In: Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy. vol. 13, p. 7. ACM New York, NY, USA (2013)

[4] Antikainen, M., Aura, T., Särelä, M.: Spook in your network: Attacking an SDN with a compromised openflow switch. In: Nordic Conference on Secure IT Systems. pp. 229–244. Springer (2014)

[5] Bifulco, R., Cui, H., Karame, G.O., Klaedtke, F.: Fingerprinting software-defined networks. In: 2015 IEEE 23rd International Conference on Network Protocols (ICNP). pp. 453–459. IEEE (2015)

[6] Dinh Ngoc, T., Bui, B., Bitchebe, S., Tchana, A., Schiavoni, V., Felber, P., Hagimont, D.: Everything You Should Know About Intel SGX Performance on Virtualized Systems. Proc. ACM Meas. Anal. Comput. Syst. **3**(1), 5:1–5:21 (Mar 2019). https://doi.org/10.1145/3322205.3311076

[7] Genkin, D., Yarom, Y.: Whack-a-Meltdown: Microarchitectural Security Games [Systems Attacks and Defenses]. IEEE Security & Privacy **19**(1), 95–98 (2021)

[8] Göttel, C., Pires, R., Rocha, I., Vaucher, S., Felber, P., Pasin, M., Schiavoni, V.: Security, performance and energy trade-offs of hardware-assisted memory protection mechanisms. In: 2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS). pp. 133–142 (2018). https://doi.org/10.1109/SRDS.2018.00024

[9] Hong, S., Xu, L., Wang, H., Gu, G.: Poisoning network visibility in software-defined networks: New attacks and countermeasures. In: Proceedings 2015 Network and Distributed System Security Symposium. Internet Society (2015). https://doi.org/10.14722/ndss.2015.23283

[10] Intel: Benefitting power and performance sleep loops (2015), https://software.intel.com/en-us/articles/benefitting-power-and-performance-sleep-loops

[11] Jacquin, L., Shaw, A., Dalton, C.: Towards trusted software-defined networks using a hardware-based integrity measurement architecture. In: Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft) (04 2015). https://doi.org/10.1109/NETSOFT.2015.7116186

[12] Kaplan, D., Powell, J., Woller, T.: AMD memory encryption. White paper, Advanced Micro Devices, Inc (April 2016)

[13] Kim, T., Park, J., Woo, J., Jeon, S., Huh, J.: ShieldStore: Shielded In-memory Key-value Storage with SGX. In: Proceedings of the Fourteenth EuroSys Conference 2019. pp. 14:1–14:15. EuroSys '19, ACM, New York, NY, USA (2019). https://doi.org/10.1145/3302424.3303951

[14] Kleinberg, J., Tardos, E.: Algorithm Design. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2005)

[15] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., et al.: Spectre attacks: Exploiting speculative execution. In: 2019 IEEE Symposium on Security and Privacy (SP). pp. 1–19. IEEE (2019)

[16] Koponen, T., Amidon, K., Balland, P., Casado, M., Chanda, A., Fulton, B., Ganichev, I., Gross, J., Ingram, P., Jackson, E., Lambeth, A., Lenglet, R., Li, S.H., Padmanabhan, A., Pettit, J., Pfaff, B., Ramanathan, R., Shenker, S., Shieh, A., Stribling, J., Thakkar, P., Wendlandt, D., Yip, A., Zhang, R.: Network virtualization in multi-tenant datacenters. In: 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14). pp. 203–216. USENIX Association, Seattle, WA (2014), https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/koponen

[17] Li, M., Zhang, Y., Lin, Z., Solihin, Y.: Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization. In: 28th USENIX Security Symposium (USENIX Security 19). pp. 1257–1272. USENIX Association, Santa Clara, CA (Aug 2019), https://www.usenix.org/conference/usenixsecurity19/presentation/li-mengyuan

[18] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., et al.: Meltdown: Reading kernel memory from user space. In: 27th USENIX Security Symposium (USENIX Security 18). pp. 973–990 (2018)

[19] Madi, T., Majumdar, S., Wang, Y., Jarraya, Y., Pourzandi, M., Wang, L.: Auditing security compliance of the virtualized infrastructure in the cloud: Application to openstack. In: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy. p. 195–206. CODASPY '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2857705.2857721

[20] McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C.V., Shafi, H., Shanbhogue, V., Savagaonkar, U.R.: Innovative Instructions and Software Model for Isolated Execution. In: Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy. pp. 10:1–10:1. HASP '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2487726.2488368

[21] Medina, J., Paladi, N., Arlos, P.: Protecting openflow using intel sgx. In: 2019 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN). pp. 1–6 (2019). https://doi.org/10.1109/NFV-SDN47374.2019.9039980

[22] Paladi, N., Gehrmann, C.: Bootstrapping trust in software defined networks. ICST Transactions on Security and Safety **4**, 153397 (12 2017). https://doi.org/10.4108/eai.7-12-2017.153397

[23] Pfaff, B., Pettit, J., Amidon, K., Casado, M., Koponen, T., Shenker, S.: Extending networking into the virtualization layer. In: Hotnets (2009)

[24] Qin, X., Jiang, H., Manzanares, A., Ruan, X., Yin, S.: Dynamic load balancing for i/o-intensive applications on clusters. ACM Transactions on Storage (TOS) **5**(3), 1–38 (2009)

[25] Schwarz, M., Gruss, D.: How Trusted Execution Environments Fuel Research on Microarchitectural Attacks. IEEE Security Privacy **18**(5), 18–27 (2020). https://doi.org/10.1109/MSEC.2020.2993896

[26] Shih, M.W., Kumar, M., Kim, T., Gavrilovska, A.: S-nfv: Securing nfv states by using sgx. In: Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks and Network Function Virtualization. p. 45–48. SDN-NFV Security '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2876019.2876032

[27] Shu, Z., Wan, J., Li, D., Lin, J., Vasilakos, A.V., Imran, M.: Security in software-defined networking: Threats and countermeasures. Mobile Networks and Applications **21**(5), 764–776 (Oct 2016). https://doi.org/10.1007/s11036-016-0676-x

[28] Svenningsson, J., Paladi, N., Vahidi, A.: Faster enclave transitions for io-intensive network applications. In: Proceedings of the ACM SIGCOMM 2021 Workshop on Secure Programmable Network INfrastructure. p. 1–8. SPIN '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3472873.3472879, https://doi.org/10.1145/3472873.3472879

[29] Svenningsson, J., Paladi, N., Vahidi, A.: Speeding up enclave transitions for IO-intensive applications (2021), https://arxiv.org/abs/2112.07339

[30] Tian, H., Zhang, Q., Yan, S., Rudnitsky, A., Shacham, L., Yariv, R., Milshten, N.: Switchless calls made practical in intel sgx. In: Proceedings of the 3rd Workshop on System Software for Trusted Execution. pp. 22–27. SysTEX '18, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3268935.3268942

[31] Wang, W., Chen, G., Pan, X., Zhang, Y., Wang, X., Bindschaedler, V., Tang, H., Gunter, C.A.: Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 2421–2434 (2017)

[32] Weichbrodt, N., Aublin, P.L., Kapitza, R.: Sgx-perf: A performance analysis tool for intel sgx enclaves. In: Proceedings of the 19th International Middleware Conference. pp. 201–213. Middleware '18, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3274808.3274824, http://doi.acm.org/10.1145/3274808.3274824

[33] Weisse, O., Bertacco, V., Austin, T.: Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves. SIGARCH Comput. Archit. News **45**(2), 81–93 (Jun 2017). https://doi.org/10.1145/3140659.3080208

[34] Yao, J., Zimmer, V.: Virtual Firmware, pp. 459–491. Apress, Berkeley, CA (2020). https://doi.org/10.1007/978-1-4842-6106-4-13

[35] Zhu, J., Hou, R., Wang, X., Wang, W., Cao, J., Zhao, B., Wang, Z., Zhang, Y., Ying, J., Zhang, L., Meng, D.: Enabling rack-scale confidential computing using heterogeneous trusted execution environment. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 1450–1465 (2020). https://doi.org/10.1109/SP40000.2020.00054